MAST 680 Assignment 3

Serly Ishkhanian

April 25, 2023

Abstract

In this paper, we will use the Lorenz equations to train one neural network that takes the solutions (the state variables) to the next and functions for three different parameter values. We will observe how we can find the best loss function, how far into the future the predictions remain accurate on the specified parameter values, the effects of manipulating the timestep between the states, and the ability of the neural network to predict the future state variables for parameter values outside the training data.

1 Introduction and Overview

From medical diagnosis to speech and image recognition, natural language processing to forecasting, neural networks are widely used in those applications. The versatility and ability of neural networks to analyze and learn the hidden complex patterns to make predictions make it applicable in many fields. By training on the data, neural networks aim to approximate functions that describe the underlying complex relationships in the data.

Using the solutions of the Lorenz equations obtained from employing three different values for one of the parameters, the task is to train one neural network that takes those state variables and predicts the next ones. We will train the neural network using the python library TensorFlow. We will determine the best loss function that provides the best predictions and the accuracy of those predictions on the defined parameter values. Additionally, we will explore the impact of adjusting the timestep between the states and the neural network's ability to predict future state variables for parameter values outside the training data. Theoretical background of neural networks is in Section 2, and implementation and results are in Sections 3 and 4, respectively.

2 Theoretical Background

Given sequential (timeseries) data, let $X \in \mathbb{R}^{M\times K}$, and $Y \in \mathbb{R}^{N\times K}$, represent the input (training) and output matrices, respectively.

$$\begin{aligned} X &= \begin{bmatrix} | & | & & | \\ X_1 & X_2 & \dots & X_K \\ | & | & & | \end{bmatrix} \\ Y &= \begin{bmatrix} | & | & & | \\ Y_1 & Y_2 & \dots & Y_K \\ | & | & & | \end{bmatrix} \end{aligned} \tag{1}$$

The aim is to find a function that continuously maps data points $X_k \mapsto Y_k$ for $k = 1, \dots, K$ where $Y_k = X_{k+1}$. How do we find this function? We can approximate it with a neural network.

Now, let's see what neural networks are. A single-layer neural network is given by a function which takes input $x \in \mathbb{R}^M$ and output $y \in \mathbb{R}^N$ as follows [1]:

$$y = \sigma(Wx + b), \tag{2}$$

where $\sigma : \mathbb{R} \mapsto \mathbb{R}$ is an activation function, $b \in \mathbb{R}^N$ is the bias of the neural network, and $W \in \mathbb{R}^{N \times M}$ is a matrix with weights as its entries. The entries of b and W are randomly initialized at the beginning, and then get adjusted during the training process. To approximate any function, a single layer might not be sufficient. Thus, (2) can be generalized to by taking compositions of the layers [1]:

$$y = \sigma_d(W_d \cdot \sigma_{d-1}(W_{d-1} \dots (W_2 \cdot \sigma_1(W_1 + b_1) + b_2) + b_{d-1}) + b_d), \tag{3}$$

where d denotes the depth of the neural network (i.e. the number of layers), each $W_j \in \mathbb{R}^{n_{j+1} \times n_j}$, and $b_j \in \mathbb{R}^{n_{j+1}}$ with $n_j \geq 1$ for $j = 1, \dots, d$ and $n_d = N$. In (3), when input x is given, it goes to the first layer through $\sigma_1(W_1 x + b_1)$. This output then becomes the input for $\sigma_2(W_2 x + b_2)$ to move to the second layer, and this process continues from one layer to the next until the final output is produced.

So, to approximate the function that would enable us to move from $X_k \mapsto Y_k$, we can do so by estimating it with a function of the form (3) with $d \geq 1$ and $n_j \geq 1$. Now, denote $g(x)$ to represent the neural network (3). We want to find the best function g such that $Y_k \approx g(X_k)$. To do so, appropriate values for the weight matrix and bias vector will be chosen during the training process that minimize the loss function.

This brings us to the loss function. The performance of the neural network can be observed from how close is the output produced by the neural network $g(X_k)$ is to the true value Y_k. This closeness is measured by the mean square error (MSE) which is the loss the function:

$$\mathcal{L} = \|Y - g(X_k)\|_{mse} = \frac{1}{2K} \sum_{k=1}^{K} \|Y_k - g(X_k)\|_2^2, \tag{4}$$

where $\| \cdot \|_2$ denotes the standard Euclidean norm for vectors. Now, the goal is to minimize the loss function $\mathcal{L}$ (i.e. minimize the difference) and in the training process, this attempted by finding weight and bias values that minimize the loss function and hence improving the neural network's performance.

Since the data is sequential, that would imply that $g(g(X_k)) = g^2(X_k) \approx X_{k+1}$. In general, $g^s(X_k) \approx X_{k+s}$. The loss function can be improved by considering this in the calculations:

$$\mathcal{L} = \frac{1}{S} \sum_{s=1}^{S} \frac{1}{2K} \sum_{k=1}^{K-s} \|X_{k+s} - g^s(X_k)\|_2^2, \quad \text{for } S \geq 1. \tag{5}$$

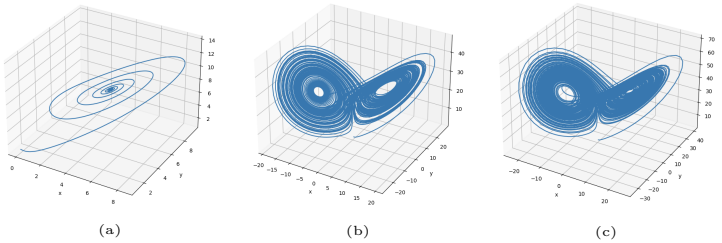
This process provides the neural network about S steps of the data into future.

3 Algorithm Implementation and Development

Before creating a model and training the neural network, first, we need to generate our data points using the Lorenz equations with $\rho = 10, 28,$ and 40.

$$\begin{aligned} \frac{dx}{dt} &= 10(y - x) \\ \frac{dy}{dt} &= x(\rho - z) - y \\ \frac{dz}{dt} &= xy - \frac{8}{3}z \end{aligned} \tag{6}$$

The Lorenz equations is a system of ordinary differential equations that exhibit chaotic dynamics. So, to generate our data points $(x(t), y(t), z(t))$, we use a built in ODE integrator **solve_ivp** in python using the initial conditions $[0, 1, 1]$. We generate 10000 data points with a timestep of 0.01. Now, for each ρ, we obtain a data matrix of size 4×10000 where the first



(a) (b) (c)

Figure 1: Plots of the Lorenz attractor for (a) $\rho = 10$, (b) $\rho = 28$, and $\rho = 40$ using the initial conditions $[0, 1, 1]$.

three rows represent the state variables (x, y, z), and the last row contains the value of ρ.

Since we are interested in training one neural network that works for $\rho = 10, 28$, and 40, we create a data matrix M by combining each of the individual data matrices obtained from each ρ. This would provide us with a data matrix M of size 4×30000. Next, we create our input matrix $X \in \mathbb{R}^{4\times 29999}$ and the output data $Y \in \mathbb{R}^{4\times 29999}$. We use X as our training data and we will compare the predictions of the neural network (the next state variables) with Y.

3.1 Finding the Loss Function

We want to train a neural network that takes a four dimensional vector $(x(t), y(t), z(t), \rho)$ as input, and outputs a three dimensional vector of the next state variables $(x(t+\Delta t), y(t+\Delta t), z(t+\Delta t))$ (in our case, $\Delta t = 0.01$). To do so, we need to find the loss function that provides the best predictions.

Using a model with 2 hidden layers of width 32 each, with a ReLU activation function and the Adam optimizer (learning rate = 0.0005), finding a loss function that provides the best predictions leads to simply comparing different loss functions. By comparing the Mean Squared Error, the Mean Absolute Error, the Mean Absolute Percentage Error, and the Huber loss functions on the model, we observe from Figure 2 that the Huber loss function had the smallest loss values implying that it would provide the best predictions compared to the other loss functions. Section B contains the code that assesses the different loss functions.

3.2 Determining Accuracy of the Predictions

To determine how far into the future do the predictions remain accurate with the trained parameter values $\rho = 10, 28$, and 40, we first crate a function that outputs a matrix containing the predictions of the network. We start by inserting the state variable at $t = 0$ into the network, then every output obtained is appended to the matrix and is fed back into the network. However, since the output is a three dimensional vector, value of ρ is added. Now, since determining value of ρ would be slightly difficult to determine, we create a prediction matrix for each ρ separately, and because of time complexity, we compute the predictions of the first 1000 state variables as shown in Algorithm 1.

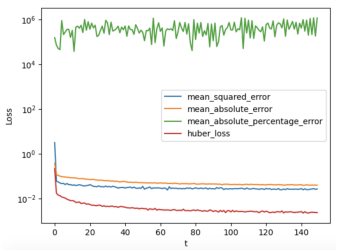


Figure 2: Plot of the results of the Mean Squared Error, the Mean Absolute Error, the Mean Absolute Percentage Error, and the Huber loss functions assessed on the neural network. Number of epochs = 150, and batch size = 10.

To determine the error between the predictions and the true values (from the Y matrix for

Algorithm 1: Create the Prediction Matrix

Create X and Y matrices for each data matrix obtained for $\rho = 10, 28$, and 40.
Create a function to produce a prediction matrix, taking input X and Y
rho = X[3, 0]
n = number of predictions desired
pred = np.zeros((n,4))
Obtain the first prediction of $t = 0$
pred1 = model.predict(X[: , 0:1].T, verbose = 0)
pred[0, :]= np.append(pred1,rho)
for $i = 1 : n$ **do**
  Feed the predicted output back into the network
  pred1 = model.predict(pred[i-1 , :].reshape((1,4)), verbose = 0)
  Append to the matrix
  pred[i , :] = np.append(pred1, rho)
end for
Return the prediction matrix

each ρ), we compute the Mean Squared Error. Lastly, to observe the effect of changing of increasing and decreasing Δt, we simply would decrease and increase the number of points generated, respectively. The reason is because since the points are generated by using *solve_ivp* on the interval $[0, 100]$, $\Delta t = \frac{100}{\text{number of data points}}$.

3.3 Parameter Values Outside the Training Set

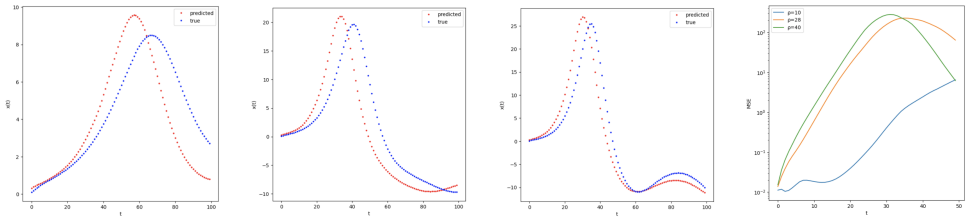
To observe how well the neural network predicts the state variables for values of $\rho = 17$, and 35, we will repeat the process above. For more details, please view the code in Section B.

4 Computational Results

Initially, I attempted training the neural network on the data points generated from using the initial conditions $[0, 1, 1.05]$. However, since the results were not satisfactory, I wanted to see if the outcomes would change by using different initial conditions. Yes, the results got better. Thus, I generated the input data using the initial conditions $[0, 1, 1]$. Regarding the learning rate, during the training process, it was observed that no matter how small I made the learning rate, the loss values were not constantly decreasing. There were small jumps. So, I decided to use 0.005 as the learning rate value.

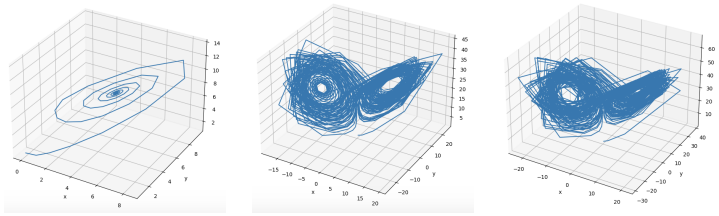
Now, let's discuss the obtained results after implementing the algorithm. To observe how far into the future the predictions of the neural network remain accurate, Figure 3 demonstrates the scatter plots of the true and predicted values of $x(t)$ of the state variables for $\rho = 10, 28,$ and 40 where $\Delta t = 0.01$. Moreover, it also provides a plot of the mean squared error between the true and predicted values. We notice that at the initial states, the $MSE \leq 10^{-1}$ for all parameters. By observing the scatter plots, we notice that for $\rho = 28$, it predicts 5 steps into future with $MSE < 10^{-1}$. For $\rho = 40$, it predicts 3 steps into future with $MSE < 10^{-1}$. Now, for $\rho = 10$, it can predict 25 steps into the future with $MSE < 10^{-1}$. However, when we observe the scatter plot, the absolute difference of the predicted and true values is within 20^{-1}. For all three parameter values, the predictions of the network get worse as as we attempt to predict further state positions and this can be seen from how large the MSE values gets.

Now, let's see how changing Δt affects the predictions of the neural network. Generating 1000 points, making $\Delta t = 0.1$, the plots of the Lorenz attractor for $\rho = 10, 28,$ and 40 are shown in the



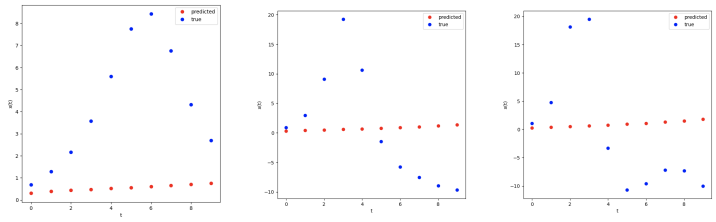
(a) (b) (c) (d)

Figure 3: Scatter plots of the true and the predicted state variables $x(t)$ for (a) $\rho = 10$, (b) $\rho = 28$, (c)$\rho = 40$ with $\Delta t = 0.01$ of the first 100 state positions, and (d) represents the MSE between the true and predict values for each ρ.



(a) (b) (c)

Figure 4: The Lorenz attractor for (a) $\rho = 10$, (b)$\rho = 28$, and (c) $\rho = 40$ using 1000 points and $\Delta t = 0.1$. Initial conditions $[0, 1, 1]$.



(a) (b) (c)

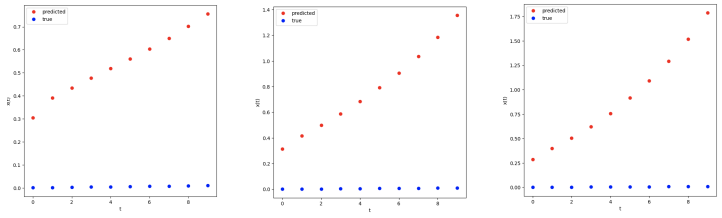
Figure 5: Scatter plot of the true and the predicted state variables (using $x(t)$) for (a) $\rho = 10$, (b) $\rho = 28$, (c)$\rho = 40$ of the first 10 state positions for $\Delta t = 0.1$.

Figure 4. Moreover, Figure 5 represents the corresponding scatter plots of $\rho = 10, 28,$ and 40. Notice that compared to when $\Delta t = 0.01$, the predictions observed in Figure 5 are worse. It seems that only the first prediction for $\rho = 28$ in Figure 5b is reasonably close to the true value.

Now, let's observe the effects of making $\Delta = 0.0001$. Figure 6 show the scatter plots of the true and predicted state variable $x(t)$ for $\rho = 10, 28,$ and 40 with $\Delta t = 0.0001$. As we can see the first prediction of $\rho = 40$ is slightly better than the ones obtained from $\rho = 10$ and $\rho = 28$. However, even though Δt got smaller, the predictions did not become better for all parameter values.

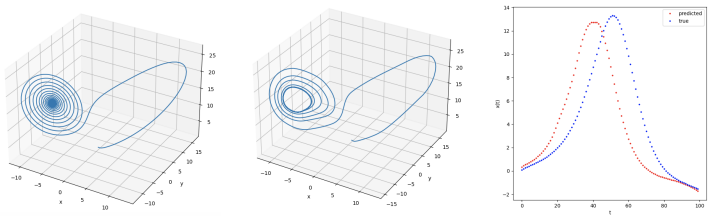
Now that we have observed the effects of changing Δt, let's experiment the performance of the neural network in predicting values for parameter values outside the training data, such as $\rho = 17,$ and 35 using $\Delta t = 0.01$. Figure 7a and 7b show the reconstructed Lorenz system of the first 1000 steps. We can see that the first few predictions are approximately the same. Not exact. Future predictions clearly get worse.

For the case when $\rho = 35$, we can see that the predictions are better than the case $\rho = 17$. However, future predictions are inaccurate.



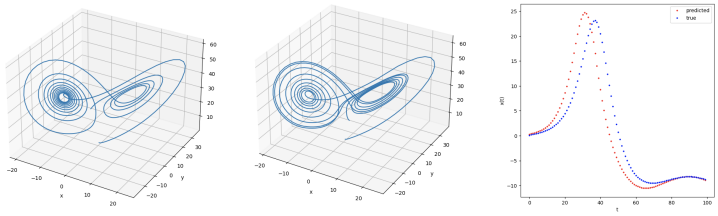
(a) (b) (c)

Figure 6: Scatter plot of the true and the predicted state variables (using $x(t)$) for (a) $\rho = 10$, (b) $\rho = 28$, (c)$\rho = 40$ with $\Delta t = 0.0001$ of the first 10 state positions.



(a) (b) (c)

Figure 7: Plot of $\rho = 17$ with $\Delta t = 0.01$ of the first 1000 steps (a) true plot (b) predicted plot, and (c) represents the scatter plot of $x(t)$.



(a) (b) (c)

Figure 8: Plot of $\rho = 35$ with $\Delta t = 0.01$ of the first 1000 steps (a) true plot (b) predicted plot, and (c) represents the scatter plot of $x(t)$.

5 Summary and Conclusions

In this report, we used the system of Lorenz equations that exhibit chaotic behaviors with $\rho = 10, 28,$ and 40 to train a neural network that works well for those three parameter values. We generate the data points by integrating the ordinary differential equations using the built-in integrator in python *solve_ivp*. We observed how we chose a loss function that provides the best predictions. We also observed how many predictions into the future the neural network provided accurately, the effects of changing Δt on the predictions, and the ability of the neural network to predict for parameter values $rho = 17$, and $\rho = 35$ for which the neural network was not trained on.

[2]

References

[1] Jason J. Bramburger. *Data-Driven Methods for Dynamical Systems.* Concordia University, 2023.

[2] N Benjamin Erichson, Lionel Mathelin, J Nathan Kutz, and Steven L Brunton. Randomized dynamic mode decomposition. *SIAM Journal on Applied Dynamical Systems*, 18(4):1867–1891, 2019.

Appendix A Python Functions

- `solve_ivp` Solves an initial value problem for a system of ODEs.
- `np.hstack()` Stacks arrays in sequence horizontally (column wise).
- `tf.keras.Sequential()` Groups a linear stack of layers into a tf.keras.Model.
- `tf.keras.optimizers.Adam()` Optimizer that implements the Adam algorithm.
- `model.compile(optimizer='...', loss=...)` Configures the model for training.
- `model.fit` Trains the model for a fixed number of epochs (dataset iterations).
- `model.predict()` Generates output predictions for the input samples.
- `model.save()` Saves keras model.
- `load_model()` Loads keras model.

Appendix B Python Code

```
import numpy as np
from scipy.integrate import solve_ivp
import tensorflow as tf
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from tensorflow.keras.losses import MeanSquaredError
from tensorflow.keras.losses import MeanAbsoluteError
from tensorflow.keras.losses import MeanAbsolutePercentageError
from tensorflow.keras.losses import Huber

from keras.models import load_model

# load the saved model
model1 = load_model('initmy_model.h5')

"""# Create the data matrix and plot the Lorenz system """

def make_points_smaller(initial_conditions, rho, sigma=10, beta=8/3, timestep=10000 ):

  WIDTH, HEIGHT, DPI = 1000, 750, 100

  # Initial conditions.

  u0, v0, w0 = initial_conditions

  # Maximum time point and total number of time points.
  tmax, n = 100, timestep

  def lorenz(t, X, sigma, beta, rho):
    """The Lorenz equations."""
    x,y,z = X
    ux = sigma*(y - x)
    uy = x*(rho-z)-y
    uz = x*y-beta*z
```

```
    return ux, uy, uz

  # Integrate the Lorenz equations.
  soln = solve_ivp(lorenz, (0, tmax), (u0, v0, w0), args=(sigma, beta, rho),
                   dense_output=True)

  # Interpolate solution onto the time grid, t.
  t = np.linspace(0, tmax, n)
  x, y, z = soln.sol(t)

  M = np.array([x,y,z, np.repeat(rho,n)])
  # Plot the Lorenz attractor using a Matplotlib 3D projection.

  WIDTH, HEIGHT, DPI = 1000, 750, 100
  fig = plt.figure(figsize=(WIDTH/DPI, HEIGHT/DPI))
  r = rho
  ax = fig.add_subplot(111, projection='3d')
  ax.plot(x, y, z)
  ax.set_xlabel('x')
  ax.set_ylabel('y')
  ax.set_zlabel('z')
  ax.set_title('Lorenz system (=%d)'%r)
  plt.show()

  return M

"""Data for rho=10,28,40"""

m10= make_points_smaller([0,1,1],10)
m28= make_points_smaller([0,1,1],28)
m40= make_points_smaller([0,1,1],40)

# Combine all the training and testing data

Train_all = np.hstack([m10,m28,m40])
Xall = Train_all[:,:-1]
Yall = Train_all[:,1:]


"""# Loop to find the best loss function """

loss_functions = [MeanSquaredError(), MeanAbsoluteError(), MeanAbsolutePercentageError(), Huber()

# Loop through each loss function
for loss_func in loss_functions:
    print('Testing', loss_func.name)
    number_of_epochs = 150
    # Create an empty list to store the loss values for each epoch
    loss_values = []
    # Loop through each epoch value

        # Define the neural network model
    model = tf.keras.models.Sequential([
      tf.keras.layers.Dense(32, activation = 'relu',input_shape= (Train_all.shape[0],)),
      tf.keras.layers.Dense(32,activation = 'relu'),
```

```
      tf.keras.layers.Dense(3)

   ])
          #Compile the model

    opt = tf.keras.optimizers.Adam(learning_rate=0.0005)


        # Compile the model with the current loss function
    model.compile(optimizer='adam', loss=loss_func)

        # Train the model with the current epoch value
    history = model.fit(Xall.T, Yall[:3,:].T,epochs=number_of_epochs,batch_size=10)

        # Add the final loss value to the list of loss values
    LOSS_LIST = list(history.history['loss'])
    loss_values.append(LOSS_LIST)

    # Plot the loss values for the current loss function
    plt.plot([i for i in range(number_of_epochs)], LOSS_LIST, label=loss_func.name)

# Add legend and labels to the plot
plt.legend()
plt.xlabel('t')
plt.ylabel('Loss')
plt.yscale('log')
plt.title('Regression Loss Functions Comparison')
plt.show()




"""# Model chosen after deciding with Huber Loss function"""

model1 = tf.keras.models.Sequential([
      tf.keras.layers.Dense(32, activation = 'relu',input_shape= (Xall.shape[0],)),
      tf.keras.layers.Dense(32,activation = 'relu'),
      tf.keras.layers.Dense(3)

])
          #Compile the model
opt = tf.keras.optimizers.Adam(learning_rate=0.0005)


        # Compile the model with the current loss function
model1.compile(optimizer='adam', loss=Huber())

        # Train the model with the current epoch value
history1 = model1.fit(Xall.T, Yall[:3,:].T,epochs=150,batch_size=10)

# train and save the model
model1.save('initmy_model.h5')
```

```
"""# Function to get prediction matrix """

def get_pred_matrix(X, Y):

  rho = X[3,0]
  num =5000
  pred_matrix = np.zeros((num,4))

  pred1 = model1.predict(X[:,0:1].T,verbose=0) # has shape(1,3)
  pred_matrix[0,:] = np.append(pred1, rho)

  for i in range(1,num):
    pred1 = model1.predict(pred_matrix[i-1,:].reshape((1,4)),verbose=0) # has shape (1,3)
    pred_matrix[i,:] = np.append(pred1, rho)

  return pred_matrix

"""# Creat X, Y matrices"""

def xy(matrix):
  x = matrix[:,:-1]
  y = matrix[:,1:]

  return x,y

"""X, Y matrices for each rho"""

x10,y10 = xy(m10)
x28,y28 = xy(m28)
x40,y40 = xy(m40)

"""Prediction matrix for each rho"""

pred_10 = get_pred_matrix(x10,y10)
pred_28 = get_pred_matrix(x28,y28)
pred_40 = get_pred_matrix(x40,y40)

"""# Function that plots the 3D plot """

def threed_plot(m,point):
  if m.shape[0]==4:
    d= m.T
  else :
    d=m


  pt=point
  WIDTH, HEIGHT, DPI = 1000, 750, 100
  fig = plt.figure(figsize=(WIDTH/DPI, HEIGHT/DPI))
  ax = fig.add_subplot(111, projection='3d')
  ax.plot(d[:pt,0:1], d[:pt,1:2], d[:pt,2:3])
  ax.set_xlabel('x')
  ax.set_ylabel('y')
  ax.set_zlabel('z')
  plt.show()
```

```
"""# Function that plots the scatter plot """

def list_plot(y,p,pt):

  fig = plt.figure(figsize=(7,7))
  plt.plot([i for i in range(pt)], p[:pt,0],'ro', markersize=2,label="predicted")
  plt.plot([i for i in range(pt)], y.T[:pt,0],'bo', markersize=2,label="true")
  plt.xlabel("t")
  plt.ylabel("x(t)")
  plt.legend()
  plt.show()

# Plot 3D of rho=10,28,40 of the predicted data
threed_plot(pred_10,1000)
threed_plot(m10,1000)

threed_plot(pred_28,1000)
threed_plot(m28,1000)

threed_plot(pred_40,1000)
threed_plot(m40,1000)


# List plot x(t) of predicted na dtrue
list_plot(y10,pred_10,100)
list_plot(y28,pred_28,100)
list_plot(y40,pred_40,100)
list_plot(y40,pred_40,10)


# Function to compute error l2-norm
def get_error_2(pred,y):

  error=[]
  s = pred.shape[0]
  r =pred[0,3]
  for i in range(s):
   l2 = np.linalg.norm(y[:,i:i+1].T-pred[i:i+1,:])



   error.append(l2)

  return error

# Function to create MSE
def get_error(pred,y):

  error=[]
  s = pred.shape[0]
  r =pred[0,3]
  for i in range(s):
   mse = np.square(np.subtract(y[:,i:i+1].T, pred[i:i+1,:])).mean()
```

```
   error.append(mse)

  return error

# To change delta, simply replace timestep with the number of data points you want to generate

# Create rho=17,35

matrix17 = make_points_smaller([0, 1, 1],17)
matrix35 = make_points_smaller([0, 1, 1],35)

x17,y17=xy(matrix17)
x35,y35=xy(matrix35)


pred_17 = get_pred_matrix(x17,y17)
pred_35 = get_pred_matrix(x35,y35)

# List plot rho=17,35

list_plot(y17,pred_17,1000)
list_plot(y17,pred_17,100)
list_plot(y17,pred_17,10)
list_plot(y35,pred_35,100)
list_plot(y35,pred_35,10)


# Compute the errors

e1 = get_error(pred_10,y10)
e28 = get_error(pred_28,y28)
e40 = get_error(pred_40,y40)
e17 = get_error(pred_17,y17)
e35 = get_error(pred_35,y35)

e1 = get_error_2(pred_10,y10)
e28 = get_error_2(pred_28,y28)

# Plot the MSE

fig = plt.figure(figsize=(7,7))
plt.plot([i for i in range(50)],e1[:50],label="=10")
plt.plot([i for i in range(50)],e28[:50],label="=28")
plt.plot([i for i in range(50)],e40[:50],label="=40")
#plt.plot([i for i in range(50)],e17[:50],label="=17")
#plt.plot([i for i in range(50)],e35[:50],label="=35")
plt.legend()
plt.ylabel("MSE")
plt.xlabel("t")
plt.yscale("log")
```